15-213
“The course that gives CMU its Zip!”

Verifying Programs with BDDs

Topics

- Representing Boolean functions with Binary Decision Diagrams
- Application to program verification
Verification Example

Do these functions produce identical results?

How could you find out?

How about exhaustive testing?

```c
int abs(int x) {
    int mask = x>>31;
    return (x ^ mask) + ~mask + 1;
}

int test_abs(int x) {
    return (x < 0) ? -x : x;
}
```
More Examples

```c
int addXY(int x, int y)
{
    return x+y;
}

int addYX(int x, int y)
{
    return y+x;
}

int mulXY(int x, int y)
{
    return x*y;
}

int mulYX(int x, int y)
{
    return y*x;
}
```
How Can We Verify Programs?

Testing
- Exhaustive testing not generally feasible
- Currently, programs only tested over small fraction of possible cases

Formal Verification
- Mathematical “proof” that code is correct

Did Pythagoras show that $a^2 + b^2 = c^2$ by testing?
Bit-Level Program Verification

```
int abs(int x) {
    int mask = x>>31;
    return (x ^ mask) + ~mask + 1;
}
```

- View computer word as 32 separate bit values
- Each output becomes Boolean function of inputs
Do these functions produce identical results?

```c
int bitOr(int x, int y) {
    return ~(~x & ~y);
}

int test_bitOr(int x, int y) {
    return x | y;
}
```

### Straight-Line Evaluation

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1 = ~x</td>
<td>v2 = ~y</td>
</tr>
<tr>
<td>v3 = v1 &amp; v2</td>
<td>v4 = ~v3</td>
</tr>
<tr>
<td>v5 = x</td>
<td>y</td>
</tr>
<tr>
<td>t = v4 == v5</td>
<td></td>
</tr>
</tbody>
</table>
Tabular Function Representation

- List every possible function value

Complexity
- Function with $n$ variables
Algebraic Function Representation

\[ f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3 \]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( x_2 \cdot x_3 \)
- \( x_1 \cdot x_3 \)

- \( f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3 \)

Boolean Algebra

Complexity

- Representation

- Determining properties of function
  - E.g., deciding whether two expressions are equivalent
Tree Representation

Truth Table

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Decision Tree

- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value

Complexity
Ordered Binary Decision Diagrams

Canonical representation of Boolean function

- Two functions equivalent if and only if graphs isomorphic
  - Can be tested in linear time
- Desirable property: *simplest form is canonical.*

Initial Tree

Reduced Graph

\[(x_1 + x_2) \cdot x_3\]
Example Functions

Constants
- 0: Unique unsatisfiable function
- 1: Unique tautology

Variable
- Treat variable as function

Typical Function
- \((x_1 + x_2) \cdot x_4\)
- No vertex labeled \(x_3\)
- Independent of \(x_3\)
- Many subgraphs shared

Odd Parity
- Linear representation
More Complex Functions

Functions
- Add 4-bit words $a$ and $b$
- Get 4-bit sum $S$
- Carry output bit $Cout$

Shared Representation
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder
- Linear growth!
Symbolic Execution

(3-bit word size)

\[ v_1 = \sim x \]

\[ v_2 = \sim y \]
Symbolic Execution (cont.)

\[ v_3 = v_1 \& v_2 \]

\[ v_4 = \lnot v_3 \]

\[ v_5 = x \mid y \]

\[ t = v_4 == v_5 \]
Find values of \( x \& y \) for which these programs produce different results

```c
int bitOr(int x, int y) {
    return ~(~x & ~y);
}

int bitXor(int x, int y) {
    return x ^ y;
}
```

### Straight-Line Evaluation

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \sim x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>( \sim y )</td>
</tr>
<tr>
<td>( v_1 = \sim x )</td>
<td>( v_2 = \sim y )</td>
</tr>
<tr>
<td>( v_3 = v_1 &amp; v_2 )</td>
<td>( v_4 = \sim v_3 )</td>
</tr>
<tr>
<td>( v_5 = x ^ y )</td>
<td>( t = v_4 == v_5 )</td>
</tr>
</tbody>
</table>
Symbolic Execution

\[ v_4 = \neg v_3 \]

\[ v_5 = x \land y \]

\[ t = v_4 \equiv v_5 \]

\[ x = 111 \]
\[ y = 001 \]
Performance: Good

```c
int addXY(int x, int y)
{
    return x+y;
}
```

```c
int addYX(int x, int y)
{
    return y+x;
}
```

[Graph showing performance comparison of Enumerate and BDD methods with Word Size on the x-axis and Seconds on the y-axis.]
**Performance: Bad**

```c
int mulXY(int x, int y)
{
    return x*y;
}
```

```c
int mulYX(int x, int y)
{
    return y*x;
}
```

---

The graph shows the performance comparison between two methods:

- **Enumerate** represents the traditional method.
- **BDD** denotes an optimized method.

The x-axis represents the **Word Size**, ranging from 0 to 32. The y-axis represents **Seconds**, ranging from 0 to 1000. The graph illustrates how the time taken increases significantly as the word size grows, particularly for the **Enumerate** method, whereas the **BDD** method remains relatively stable.
Why Is Multiplication Slow?

Multiplication function intractable for BDDs

- Exponential growth, regardless of variable ordering

<table>
<thead>
<tr>
<th>Bits</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>21</td>
<td>155</td>
</tr>
<tr>
<td>8</td>
<td>41</td>
<td>14560</td>
</tr>
</tbody>
</table>
What if Multiplication were Easy?

```c
int factorK(int x, int y) {
    int K = XXXX...X;
    int rangeOK =
        1 < x && x <= y;
    int factorOK =
        x*y == K;
    return
        !(rangeOK && factorOK);
}
```

```c
int one(int x, int y) {
    return 1;
}
```
Dealing with Conditionals

During Evaluation, Keep Track of:

- **Current Context**: Under what condition would code be evaluated
- **Definedness** (for each variable)
  - Has it been assigned a value

```c
int abs(int x)
{
    int r;
    if (x < 0)
        r = -x;
    else
        r = x;
    return r;
}
```

<table>
<thead>
<tr>
<th>Context</th>
<th>Defined</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t1 = x &lt; 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>v1 = -x</td>
<td>t1</td>
<td>0</td>
</tr>
<tr>
<td>r = v1</td>
<td>t1</td>
<td>t1</td>
</tr>
<tr>
<td>r = x</td>
<td>!t1</td>
<td>1</td>
</tr>
<tr>
<td>v2 = r</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Dealing with Loops

**Unroll**

- Turn into bounded sequence of conditionals
  - Default limit = 33
- Signal runtime error if don’t complete within limit

```c
int ilog2(unsigned x)
{
    int r = -1;
    while (x) {
        r++; x >>= 1;
    }
    return r;
}
```

**Unrolled**

```c
int ilog2(unsigned x)
{
    int r = -1;
    if (x) {
        r++; x >>= 1;
    }
    else return r;
    if (x) {
        r++; x >>= 1;
    }
    else return r;
    . . .
    if (x) {
        r++; x >>= 1;
    }
    else return r;
    error();
}
```
Evaluation

Strengths

- Provides 100% guarantee of correctness
- Performance very good for simple arithmetic functions

Weaknesses

- Important integer functions have exponential blowup
- Not practical for programs that build and operate on large data structures
Some History

Origins

- Lee 1959, Akers 1976
  - Idea of representing Boolean function as BDD
- Hopcroft, Fortune, Schmidt 1978
  - Recognized that ordered BDDs were like finite state machines
  - Polynomial algorithm for equivalence
- Bryant 1986
  - Proposed as useful data structure + efficient algorithms
- McMillan 1987
  - Developed symbolic model checking
  - Method for verifying complex sequential systems
- Bryant 1991
  - Proved that multiplication has exponential BDD
  - No matter how variables are ordered